Chapter 5

Shared Memory

5.1 Introduction

In distributed computing, various different models exist. So far, the focus of the course was on loosely-coupled distributed systems such as the Internet, where nodes asynchronously communicate by exchanging messages. The "opposite" model is a tightly-coupled parallel computer where nodes access a common memory totally synchronously—in distributed computing such a system is called a Parallel Random Access Machine (PRAM).

A third major model is somehow between these two extremes, the *shared memory* model. In a shared memory system, asynchronous processes (or processors) communicate via a common memory area of shared variables or registers:

Definition 5.1 (Shared Memory)**.** *A shared memory system is a system that consists of asynchronous processes that access a common (shared) memory. A process can atomically access a register in the shared memory through a set of predefined operations. An atomic modification appears to the rest of the system instantaneously. Apart from this shared memory, processes can also have some local (private) memory.*

Remarks:

- Various shared memory systems exist. A main difference is how they allow processes to access the shared memory. All systems can atomically read or write a shared register R. Most systems do allow for advanced *atomic* read-modify-write (RMW) operations, for example:
  - test-and-set(R): $t := R$; $R := 1$; return t
  - fetch-and-add(R, x): $t := R$; $R := R + x$; return t
  - compare-and-swap(R, x, y): if $R = x$ then $R := y$; return **true**; else return **false**; endif;
  - load-link(R)/store-conditional(R, x): Load-link returns the current value of the specified register R. A subsequent store-conditional to the same register will store a new value x (and return **true**) only if no updates have occurred to that register since the load-link. If any updates have occurred, the store-conditional is guaranteed to fail

(and return **false**), even if the value read by the load-link has since been restored.

- The power of RMW operations can be measured with the so-called *consensus-number*: The consensus-number k of a RMW operation defines whether one can solve consensus for k processes. Test-and-set for instance has consensus-number 2 (one can solve consensus with 2 processes, but not 3), whereas the consensus-number of compare-and-swap is infinite. It can be shown that the power of a shared memory system is determined by the consensus-number ("universality of consensus".) This insight has a remarkable theoretical and practical impact. In practice for instance, after this was known, hardware designers stopped developing shared memory systems supporting weak RMW operations.

- Many of the results derived in the message passing model have an equivalent in the shared memory model. Consensus for instance is traditionally studied in the shared memory model.

- Whereas programming a message passing system is rather tricky (in particular if fault-tolerance has to be integrated), programming a shared memory system is generally considered easier, as programmers are given access to global variables directly and do not need to worry about exchanging messages correctly. Because of this, even distributed systems which physically communicate by exchanging messages can often be programmed through a shared memory middleware, making the programmer's life easier.

- We will most likely find the general spirit of shared memory systems in upcoming multi-core architectures. As for programming style, the multi-core community seems to favor an accelerated version of shared memory, *transactional memory*.

- From a message passing perspective, the shared memory model is like a bipartite graph: On one side you have the processes (the nodes) which pretty much behave like nodes in the message passing model (asynchronous, maybe failures). On the other side you have the shared registers, which just work perfectly (no failures, no delay).

5.2 Mutual Exclusion

A classic problem in shared memory systems is mutual exclusion. We are given a number of processes which occasionally need to access the same resource. The resource may be a shared variable, or a more general object such as a data structure or a shared printer. The catch is that only one process at the time is allowed to access the resource. More formally:

Definition 5.2 (Mutual Exclusion)**.** *We are given a number of processes, each executing the following code sections:*
<Entry> → <Critical Section> → <Exit> → <Remaining Code>
A mutual exclusion algorithm consists of code for entry and exit sections, such that the following holds

- *Mutual Exclusion: At all times* at most one *process is in the critical section.*
- *No deadlock: If some process manages to get to the entry section, later* some *(possibly different) process will get to the critical section.*

Sometimes we in addition ask for

- *No lockout: If some process manages to get to the entry section, later* the same *process will get to the critical section.*
- *Unobstructed exit: No process can get stuck in the exit section.*

Using RMW primitives one can build mutual exclusion algorithms quite easily. Algorithm 23 shows an example with the test-and-set primitive.

Algorithm 23 Mutual Exclusion: Test-and-Set

Input: Shared register $R := 0$

<**Entry**>

1: **repeat**
2: $r :=$ test-and-set(R)
3: **until** $r = 0$

<**Critical Section**>

4: ...

<**Exit**>

5: $R := 0$

<**Remainder Code**>

6: ...

Theorem 5.3. *Algorithm 23 solves the mutual exclusion problem as in Definition 5.2.*

Proof. Mutual exclusion follows directly from the test-and-set definition: Initially R is 0. Let p_i be the i^{th} process to successfully execute the test-and-set, where successfully means that the result of the test-and-set is 0. This happens at time t_i. At time t_i' process p_i resets the shared register R to 0. Between t_i and t_i' no other process can successfully test-and-set, hence no other process can enter the critical section concurrently.

Proving no deadlock works similar: One of the processes loitering in the entry section will successfully test-and-set as soon as the process in the critical section exited.

Since the exit section only consists of a single instruction (no potential infinite loops) we have unobstructed exit. □

Remarks:

- No lockout, on the other hand, is not given by this algorithm. Even with only two processes there are asynchronous executions where always the same process wins the test-and-set.
- Algorithm 23 can be adapted to guarantee fairness (no lockout), essentially by ordering the processes in the entry section in a queue.

- A natural question is whether one can achieve mutual exclusion with only reads and writes, that is without advanced RMW operations. The answer is yes!

Our read/write mutual exclusion algorithm is for two processes p_0 and p_1 only. In the remarks we discuss how it can be extended. The general idea is that process p_i has to mark its desire to enter the critical section in a "want" register W_i by setting $W_i := 1$. Only if the other process is not interested ($W_{1-i} = 0$) access is granted. This however is too simple since we may run into a deadlock. This deadlock (and at the same time also lockout) is resolved by adding a priority variable Π. See Algorithm 24.

Algorithm 24 Mutual Exclusion: Peterson's Algorithm

Initialization: Shared registers W_0, W_1, Π, all initially 0.
Code for process p_i , $i = \{0, 1\}$
<**Entry**>
1: $W_i := 1$
2: $\Pi := 1 - i$
3: **repeat until** $\Pi = i$ or $W_{1-i} = 0$
<**Critical Section**>
4: ...
<**Exit**>
5: $W_i := 0$
<**Remainder Code**>
6: ...

Remarks:

- Note that line 3 in Algorithm 24 represents a "spinlock" or "busy-wait", similarly to the lines 1-3 in Algorithm 23.

Theorem 5.4. *Algorithm 24 solves the mutual exclusion problem as in Definition 5.2.*

Proof. The shared variable Π elegantly grants priority to the process that passes line 2 first. If both processes are competing, only process p_Π can access the critical section because of Π. The other process $p_{1-\Pi}$ cannot access the critical section because $W_\Pi = 1$ (and $\Pi \neq 1 - \Pi$). The only other reason to access the critical section is because the other process is in the remainder code (that is, not interested). This proves mutual exclusion!

No deadlock comes directly with Π: Process p_Π gets direct access to the critical section, no matter what the other process does.

Since the exit section only consists of a single instruction (no potential infinite loops) we have unobstructed exit.

Thanks to the shared variable Π also no lockout (fairness) is achieved: If a process p_i loses against its competitor p_{1-i} in line 2, it will have to wait until the competitor resets $W_{1-i} := 0$ in the exit section. If process p_i is unlucky it will not check $W_{1-i} = 0$ early enough before process p_{1-i} sets $W_{1-i} := 1$ again in line 1. However, as soon as p_{1-i} hits line 2, process p_i gets the priority due to Π, and can enter the critical section. □

Remarks:

- Extending Peterson's Algorithm to more than 2 processes can be done by a tournament tree, like in tennis. With n processes every process needs to win $\log n$ matches before it can enter the critical section. More precisely, each process starts at the bottom level of a binary tree, and proceeds to the parent level if winning. Once winning the root of the tree it can enter the critical section. Thanks to the priority variables Π at each node of the binary tree, we inherit all the properties of Definition 5.2.

5.3 Store & Collect

5.3.1 Problem Definition

In this section, we will look at a second shared memory problem that has an elegant solution. Informally, the problem can be stated as follows. There are n processes $p_1, \ldots, p_n$. Every process p_i has a read/write register R_i in the shared memory where it can *store* some information that is destined for the other processes. Further, there is an operation by which a process can *collect* (i.e., read) the values of all the processes that stored some value in their register.

We say that an operation *op1* precedes an operation *op2* iff *op1* terminates before *op2* starts. An operation *op2* follows an operation *op1* iff *op1* precedes *op2*.

Definition 5.5 (Collect)**.** *There are two operations: A* STORE(*val*) *by process p_i sets val to be the latest value of its register R_i. A* COLLECT *operation returns a* view*, a partial function V from the set of processes to a set of values, where $V(p_i)$ is the latest value stored by p_i, for each process p_i. For a* COLLECT *operation* cop*, the following validity properties must hold for every process p_i:*

- *If $V(p_i) = \perp$, then no* STORE *operation by p_i precedes* cop*.*
- *If $V(p_i) = v \neq \perp$, then v is the value of a* STORE *operation* sop *of p_i that does not follow* cop*, and there is no* STORE *operation by p_i that follows* sop *and precedes* cop*.*

Hence, a COLLECT operation *cop* should not read from the future or miss a preceding STORE operation *sop*.

We assume that the read/write register R_i of every process p_i is initialized to $\perp$. We define the step complexity of an operation *op* to be the number of accesses to registers in the shared memory. There is a trivial solution to the *collect* problem as shown by Algorithm 25.

Algorithm 25 Collect: Simple (Non-Adaptive) Solution

Operation STORE(*val*) (by process p_i) **:**
1: $R_i := val$
Operation COLLECT**:**
2: **for** $i := 1$ **to** n **do**
3: $V(p_i) := R_i$ // *read register* R_i
4: **end for**

Remarks:

- Algorithm 25 clearly works. The step complexity of every STORE operation is 1, the step complexity of a COLLECT operation is n.
- At first sight, the step complexities of Algorithm 25 seem optimal. Because there are n processes, there clearly are cases in which a COLLECT operation needs to read all n registers. However, there are also scenarios in which the step complexity of the COLLECT operation seems very costly. Assume that there are only two processes p_i and p_j that have stored a value in their registers R_i and R_j. In this case, a COLLECT in principle only needs to read the registers R_i and R_j and can ignore all the other registers.
- Assume that up to a certain time t, $k \leq n$ processes have finished or started at least one operation. We call an operation *op* at time t *adaptive* to contention if the step complexity of *op* only depends on k and is independent of n.
- In the following, we will see how to implement adaptive versions of STORE and COLLECT.

5.3.2 Splitters

Algorithm 26 Splitter Code

Shared Registers: $X : \{\perp\} \cup \{1, \ldots, n\}$; Y : **boolean**
Initialization: $X := \perp$; $Y :=$ **false**

Splitter access by process p_i:

```
1:  X := i;
2:  if Y then
3:    return right
4:  else
5:    Y := true
6:    if X = i then
7:      return stop
8:    else
9:      return left
10:   end if
11: end if
```

To obtain adaptive collect algorithms, we need a synchronization primitive, called a *splitter*.

Definition 5.6 (Splitter)**.** *A splitter is a synchronization primitive with the following characteristic. A process entering a splitter exits with either **stop**, **left**, or **right**. If k processes enter a splitter, at most one process exits with **stop** and at most $k-1$ processes exit with **left** and **right**, respectively.*

Hence, it is guaranteed that if a single process enters the splitter, then it obtains **stop**, and if two or more processes enter the splitter, then there is at most one process obtaining **stop** and there are two processes that obtain

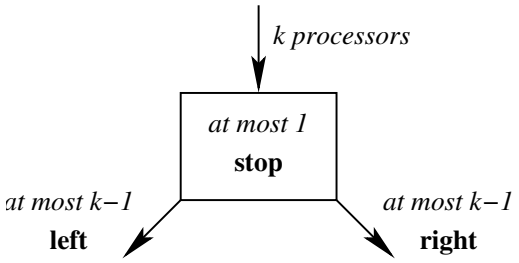

Figure 5.1: A Splitter

different values (i.e., either there is exactly one **stop** or there is at least one **left** and at least one **right**). For an illustration, see Figure 5.1. The code implementing a splitter is given by Algorithm 26.

Lemma 5.7. *Algorithm 26 correctly implements a splitter.*

Proof. Assume that k processes enter the splitter. Because the first process that checks whether $Y =$ **true** in line 2 will find that $Y =$ **false**, not all processes return **right**. Next, assume that i is the last process that sets $X := i$. If i does not return **right**, it will find $X = i$ in line 6 and therefore return **stop**. Hence, there is always a process that does not return **left**. It remains to show that at most 1 process returns **stop**. For the sake of contradiction, assume p_i and p_j are two processes that return **stop** and assume that p_i sets $X := i$ before p_j sets $X := j$. Both processes need to check whether Y is **true** before one of them sets $Y :=$ **true**. Hence, they both complete the assignment in line 1 before the first one of them checks the value of X in line 6. Hence, by the time p_i arrives at line 6, $X \neq i$ (p_j and maybe some other processes have overwritten X by then). Therefore, p_i does not return **stop** and we get a contradiction to the assumption that both p_i and p_j return **stop**. □

5.3.3 Binary Splitter Tree

Assume that we are given $2^n - 1$ splitters and that for every splitter S, there is an additional shared variable $Z_S : \{\perp\} \cup \{1, \ldots, n\}$ that is initialized to $\perp$ and an additional shared variable $M_S :$ **boolean** that is initialized to **false**. We call a splitter S marked if $M_S =$ **true**. The $2^n - 1$ splitters are arranged in a complete binary tree of height $n - 1$. Let $S(v)$ be the splitter associated with a node v of the binary tree. The STORE and COLLECT operations are given by Algorithm 27.

Theorem 5.8. *Algorithm 27 correctly implements* STORE *and* COLLECT. *Let k be the number of participating processes. The step complexity of the first* STORE *of a process p_i is $\mathcal{O}(k)$, the step complexity of every additional* STORE *of p_i is $\mathcal{O}(1)$, and the step complexity of* COLLECT *is $\mathcal{O}(k)$.*

Proof. Because at most one process can stop at a splitter, it is sufficient to show that every process stops at some splitter at depth at most $k - 1 \leq n - 1$ when invoking the first STORE operation to prove correctness. We prove that at most $k - i$ processes enter a subtree at depth i (i.e., a subtree where the root has distance i to the root of the whole tree). By definition of k, the number of

Algorithm 27 Adaptive Collect: Binary Tree Algorithm

Operation STORE(val) (by process p_i) **:**

```
1:  R_i := val
2:  if first STORE operation by p_i then
3:    v := root node of binary tree
4:    α := result of entering splitter S(v);
5:    M_S(v) := true
6:    while α ≠ stop do
7:      if α = left then
8:        v := left child of v
9:      else
10:       v := right child of v
11:     end if
12:     α := result of entering splitter S(v);
13:     M_S(v) := true
14:   end while
15:   Z_S(v) := i
16: end if
```

Operation COLLECT**:**

Traverse marked part of binary tree:

```
17: for all marked splitters S do
18:   if Z_S ≠ ⊥ then
19:     i := Z_S; V(p_i) := R_i                    // read value of process p_i
20:   end if
21: end for                                        // V(p_i) = ⊥ for all other processes
```

processes entering the splitter at depth 0 (i.e., at the root of the binary tree) is k. For $i > 1$, the claim follows by induction because of the at most $k - i$ processes entering the splitter at the root of a depth i subtree, at most $k-i-1$ obtain **left** and **right**, respectively. Hence, at the latest when reaching depth $k - 1$, a process is the only process entering a splitter and thus obtains **stop**. It thus also follows that the step complexity of the first invocation of STORE is $\mathcal{O}(k)$.

To show that the step complexity of COLLECT is $\mathcal{O}(k)$, we first observe that the marked nodes of the binary tree are connected, and therefore can be traversed by only reading the variables M_S associated to them and their neighbors. Hence, showing that at most $2k - 1$ nodes of the binary tree are marked is sufficient. Let x_k be the maximum number of marked nodes in a tree, where k processes access the root. We claim that $x_k \leq 2k - 1$, which is true for $k = 1$ because a single process entering a splitter will always compute **stop**. Now assume the inequality holds for $1, \ldots, k - 1$. Not all k processes may exit the splitter with **left** (or **right**), i.e., $k_l \leq k - 1$ processes will turn left and $k_r \leq \min\{k - k_l, k - 1\}$ turn right. The left and right children of the root are the roots of their subtrees, hence the induction hypothesis yields

$$x_k = x_{k_l} + x_{k_r} + 1 \leq (2k_l - 1) + (2k_r - 1) + 1 \leq 2k - 1,$$

concluding induction and proof. □

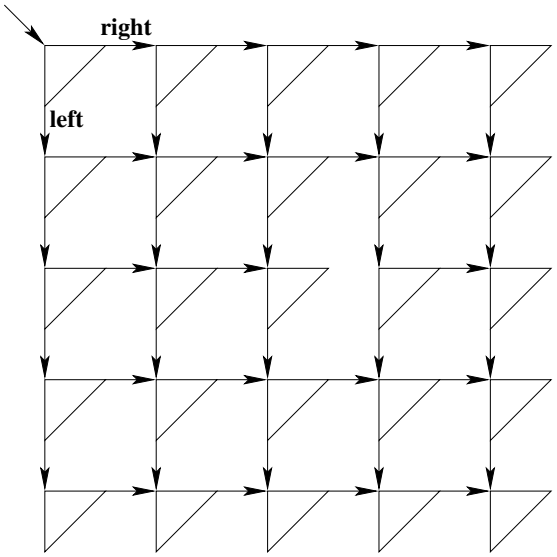

Figure 5.2: 5×5 Splitter Matrix

Remarks:

- The step complexities of Algorithm 27 are very good. Clearly, the step complexity of the COLLECT operation is asymptotically optimal. In order for the algorithm to work, we however need to allocate the memory for the complete binary tree of depth $n-1$. The space complexity of Algorithm 27 therefore is exponential in n. We will next see how to obtain a polynomial space complexity at the cost of a worse COLLECT step complexity.

5.3.4 Splitter Matrix

Instead of arranging splitters in a binary tree, we arrange n^2 splitters in an $n \times n$ matrix as shown in Figure 5.2. The algorithm is analogous to Algorithm 27. The matrix is entered at the top left. If a process receives **left**, it next visits the splitter in the next row of the same column. If a process receives **right**, it next visits the splitter in the next column of the same row. Clearly, the space complexity of this algorithm is $\mathcal{O}(n^2)$. The following theorem gives bounds on the step complexities of STORE and COLLECT.

Theorem 5.9. *Let k be the number of participating processes. The step complexity of the first* STORE *of a process p_i is $\mathcal{O}(k)$, the step complexity of every additional* STORE *of p_i is $\mathcal{O}(1)$, and the step complexity of* COLLECT *is $\mathcal{O}(k^2)$.*

Proof. Let the top row be row 0 and the left-most column be column 0. Let x_i be the number of processes entering a splitter in row i. By induction on i, we show that $x_i \leq k - i$. Clearly, $x_0 \leq k$. Let us therefore consider the case $i > 0$. Let j be the largest column such that at least one process visits the splitter in row $i-1$ and column j. By the properties of splitters, not all processes entering the splitter in row $i-1$ and column j obtain **left**. Therefore, not all processes entering a splitter in row $i-1$ move on to row i. Because at least one process

stays in every row, we get that $x_i \leq k - i$. Similarly, the number of processes entering column j is at most $k - j$. Hence, every process stops at the latest in row $k - 1$ and column $k - 1$ and the number of marked splitters is at most k^2. Thus, the step complexity of COLLECT is at most $\mathcal{O}(k^2)$. Because the longest path in the splitter matrix is $2k$, the step complexity of STORE is $\mathcal{O}(k)$. □

Remarks:

- With a slightly more complicated argument, it is possible to show that the number of processes entering the splitter in row i and column j is at most $k - i - j$. Hence, it suffices to only allocate the upper left half (including the diagonal) of the $n \times n$ matrix of splitters.
- The binary tree algorithm can be made space efficient by using a randomized version of a splitter. Whenever returning left or right, a randomized splitter returns left or right with probability 1/2. With high probability, it then suffices to allocate a binary tree of depth $\mathcal{O}(\log n)$.
- Recently, it has been shown that with a considerably more complicated deterministic algorithm, it is possible to achieve $\mathcal{O}(k)$ step complexity and $\mathcal{O}(n^2)$ space complexity.

Chapter Notes

Already in 1965 Edsger Dijkstra gave a deadlock-free solution for mutual exclusion [Dij65]. Later, Maurice Herlihy suggested consensus-numbers [Her91], where he proved the "universality of consensus", i.e., the power of a shared memory system is determined by the consensus-number. For this work, Maurice Herlihy was awarded the Dijkstra Prize in Distributed Computing in 2003. Petersons Algorithm is due to [PF77, Pet81], and adaptive collect was studied in the sequence of papers [MA95, AFG02, AL05, AKP$^+$06].

Bibliography

[AFG02] Hagit Attiya, Arie Fouren, and Eli Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

[AKP$^+$06] Hagit Attiya, Fabian Kuhn, C. Greg Plaxton, Mirjam Wattenhofer, and Roger Wattenhofer. Efficient adaptive collect using randomization. *Distributed Computing*, 18(3):179–188, 2006.

[AL05] Yehuda Afek and Yaron De Levie. Space and Step Complexity Efficient Adaptive Collect. In *DISC*, pages 384–398, 2005.

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[Her91] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[MA95] Mark Moir and James H. Anderson. Wait-Free Algorithms for Fast, Long-Lived Renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995.

[Pet81] J.L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.

[PF77] G.L. Peterson and M.J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 91–97. ACM, 1977.